Spring Boot 对测试的支持

1. 开发人员编码需要做的单元测试
2. 微服务和微服务之间的接口联调测试
3. 最后是微服务和微服务之间的集成测试

- 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

为了可以在测试中获取到启动后的上下文环境（Beans），Spring Boot Test 提供了两个注解来支持。测试时只需在测试类的上面添加 @RunWith(SpringRunner.class) 和 @SpringBootTest 注解即可。

证明测试服务成功。但是这种测试会稍显麻烦，因为控制台打印了太多的东西，需要我们来仔细分辨，这里有更优雅的解决方案，可以利用 OutputCapture 来判断 System 是否输出了我们想要的内容.

```
import static org.assertj.core.api.Assertions.assertThat;
import org.springframework.boot.test.rule.OutputCapture;

@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloServiceTest {
    @Rule
    public OutputCapture outputCapture = new OutputCapture();
    @Resource
    HelloService helloService;
    @Test
    public void  sayHelloTest(){
        helloService.sayHello();
        assertThat(this.outputCapture.toString().contains("hello service")).isTrue(
);
    }
}
```

OutputCapture 是 Spring Boot 提供的一个测试类，它能捕获 System.out 和 System.err 的输

出，我们可以利用这个特性来判断程序中的输出是否执行。

Web 测试

Spring Boot Test 中有准对 Web 测试的解决方案：MockMvc，MockMvc 实现了对 Http 请求的模拟，能够直接使用网络的形式，转换到 Controller 的调用，这样可以使得测试速度更快、不依赖网络环境，而且提供了一套验证的工具，这样可以使得请求的验证统一而且更方便。

```
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.pri
nt;
@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloWebTest {
    private MockMvc mockMvc;
    @Before
    public void setUp() throws Exception {
        mockMvc = MockMvcBuilders.standaloneSetup(new HelloController()).build();
    }
    @Test
    public void testHello() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post("/hello")
                .accept(MediaType.APPLICATION_JSON_UTF8)).andDo(print());
    }
}
```

- @Before 注意意味着在测试用例执行前需要执行的操作，这里是初始化需要建立的测试环境。
- MockMvcRequestBuilders.post 是指支持 post 请求，这里其实可以支持各种类型的请求，比如 get 请求、put 请求、patch 请求、delete 请求等。
- andDo(print())，andDo()：添加 ResultHandler 结果处理器，print() 打印出请求和相应的内容

但有时候我们并不想知道整个请求流程，只需要验证返回的结果是否正确即可，我们可以做下面的改造：

```
@Test
public void testHello() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post("/hello")
            .accept(MediaType.APPLICATION_JSON_UTF8))
//                .andDo(print())
             .andExpect(content().string(equalTo("hello web")));
}
```

如果接口返回值是"hello web"测试执行成功，否则测试用例执行失败。也支持验证结果集中是否包含了特定的字符串，这时可以使用 containsString() 方法来判断。

`.andExpect(content().string(containsString("hello")));`

支持直接将结果集转换为字符串输出：

```
String mvcResult= mockMvc.perform(MockMvcRequestBuilders.get("/messages")).andRetur
n().getResponse().getContentAsString();
System.out.println("Result === "+mvcResult);
```

支持在请求的时候传递参数：

```
@Test
public void testHelloMore() throws Exception {
    final MultiValueMap<String, String> params = new LinkedMultiValueMap<>();
    params.add("id", "6");
    params.add("hello", "world");
    mockMvc.perform(
             MockMvcRequestBuilders.post("/hello")
            .params(params)
            .contentType(MediaType.APPLICATION_JSON_UTF8)
            .accept(MediaType.APPLICATION_JSON_UTF8))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("hello")));;
}
```

返回结果如果是 Json 可以使用下面语法来判断：

```
.andExpect(MockMvcResultMatchers.jsonPath("$.name").value("纯洁的微笑"))
```

- perform 构建一个请求，并且返回 ResultActions 实例，该实例则可以获取到请求的返回内容。
- params 构建请求时候的参数，也支持 param(key,value) 的方式连续添加。
- contentType(MediaType.APPLICATION_JSON_UTF8) 代表发送端发送的数据格式。
- accept(MediaType.APPLICATION_JSON_UTF8) 代表客户端希望接受的数据类型格式。
- mockMvc.perform() 建立 Web 请求。
- andExpect(...) 可以在 perform(...) 函数调用后多次调用，表示对多个条件的判断。
- status().isOk() 判断请求状态是否返回 200。
- andReturn 该方法返回 MvcResult 对象，该对象可以获取到返回的视图名称、返回的 Response 状态、获取拦截请求的拦截器集合等。

JUnit 使用

Junit 常用注解

- @Test：把一个方法标记为测试方法
- @Before：每一个测试方法执行前自动调用一次
- @After：每一个测试方法执行完自动调用一次
- @BeforeClass：所有测试方法执行前执行一次，在测试类还没有实例化就已经被加载，所以用 static 修饰
- @AfterClass：所有测试方法执行完执行一次，在测试类还没有实例化就已经被加载，所以用 static 修饰
- @Ignore：暂不执行该测试方法
- @RunWith 当一个类用 @RunWith 注释或继承一个用 @RunWith 注释的类时，JUnit 将调用它所引用的类来运行该类中的测试而不是开发者去在 junit 内部去构建它。我们在开发过程中使用这个特性。

Assert 使用

它断定某一个实际的运行值和预期想一样，否则就抛出异常 。

```
//验证结果是否为空
        Assert.assertNotNull(userService.getUser());
        //验证结果是否相等
        Assert.assertEquals("i am neo!", userService.getUser());
        //验证条件是否成立
        Assert.assertFalse(1+1>3);
        //验证对象是否相等
        Assert.assertSame(userService,userService);
        int status=404;
        //验证结果集，提示
        Assert.assertFalse("错误，正确的返回值为200", status != 200);
        String[] expectedOutput = {"one", "two", "one"};
        String[] methodOutput = {"one", "one", "two"};
        //验证数组是否相同
        Assert.assertArrayEquals(expectedOutput, methodOutput);
```

Junt 使用的几条建议：

- 测试方法上必须使用 @Test 进行修饰
- 测试方法必须使用 public void 进行修饰，不能带任何的参数
- 新建一个源代码目录来存放我们的测试代码，即将测试代码和项目业务代码分开

- 测试类所在的包名应该和被测试类所在的包名保持一致
- 测试单元中的每个方法必须可以独立测试，测试方法间不能有任何的依赖
- 测试类使用 Test 作为类名的后缀（不是必须）
- 测试方法使用 test 作为方法名的前缀（不是必须）